Data Management in Large-Scale Distributed Systems Project

MOSIG M2 - 2019/20

Members: Gabriel Benevides - Gabriel ANTHUNES JOB

Project Description

In this project we will carry on several analysises on the data set available online representing 29 days worth of information on one of *Google*'s large scale clusters (https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md), compromising about 12.5k machines. There are hundreds of files available detailing the behavior of said machines, as well as the jobs and tasks they are carrying out over this time period.

The data were downloaded using a custom made script, **downloads.sh**, which was written to automatically push files from google's cloud storage (using the *GSUtil* (https://cloud.google.com/storage/docs/gsutil) tool) and unzip them immediately after in the desired path destination. To facilitate the final calculations, we have decided to not use **all** the files, instead focusing on at most **100** files for each category. That is, when that many are available. Each analysis will be made using *spark* transformations on the resilient distributed dataset (RDD) created for manipulation of out data.

In addition to this *report* the files in the submission are:

- One *python* (.py) file for each of the analyses or questions answered.
- The aforementioned custom-made script **script.sh**, which helped downloading and unzipping the task/job events files.
- Images containing the graphics shown in the report, in the *images* folder.
- The *allResults.txt* file, which has a record of the output of each of the anaylises, for consultation purposes.

Work on the Dataset

The studies executed where made to show a few possibilities of what **Spark** can do. With that said, the analyses will consist of basic questions, some suggested in the *project description* (https://tropars.github.io/downloads/lectures/LSDM/LSDM-lab-spark-google.pdf) and some which were not. In some cases, the *time* library of python will be used also to register the elapsed time for a given processing to be done by spark. Once again, it is worth mentioning that the computational resources were limited. As such, we decided to use only *100 files for task and job events*, out of 500 in total. This number would not slow the machines down so much, as well as not cost so much space in the hard-drive, while still being able to measure the capabilities of Spark.

We move on to the anaylises.

Analysis of CPU loss due to maintenance

In this step we are interested in estimating how much CPU is lost due to maintenance. To do so, we will process data on the *machine_events* files. We will need first to set an RDD with the amount of cpu removed during the processing. This information would take a new form to look like this:

```
(machine_id, (event_type, cpu, 1))
```

The one value at the end will be used posteriously in our calculations.

The code to reuce the original *RDD* to this structure is a bit complex and looks like this:

```
cpu_removed = entries.filter(lambda x: x[2] != u'2' and x[4] is not u'')\
    .map(lambda x:(x[1],(int(x[2]),float(x[4]),1)) if x[2]==u'1' else (x[1],(int(x[2]),0
,1)))\
    .reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1], x[2]+y[2]))\
    .filter(lambda x: x[1][0] != 0);
```

Following these operations we need to compute the amount of cpu lost for all removals represented in the form: `(nb of removals, cpu loss)` The code to reduce our *RDD* to this form looks like this:

```
cpu_loss_all = cpu_removed.map(lambda x: (x[1][0], x[1][1])).reduce(lambda x,y: (x[0] +
 y[0], x[1] + y[1]))
```

Finally we can extract the total number of maintenances and the amount of cpu lost by them, which finally enables us to estimate the percentages that we want:

```
cpu_loss_maintenance = cpu_removed.map(lambda x: (x[1][0], x[1][1]) if 2*x[1][0]+1 == x[
1][2] else (x[1][0], x[1][1] - x[1][1]/x[1][0])).filter(lambda x: x[0] > 0).reduce(lambd
a x,y: (x[0] + y[0], x[1] + y[1]))

tot_cpu_loss_all = cpu_loss_all[1] / cpu_loss_all[0] * 100

tot_cpu_loss_mtnc = cpu_loss_maintenance[1] / cpu_loss_maintenance[0] * 100

tot_cpu_loss_fail = tot_cpu_loss_all - tot_cpu_loss_mtnc
```

The same procedure can be repeated to extract information related to memory loss. And finally, the results are as follows:

```
The percentage of cpu loss for all removals is 53.187451155520826
The percentage of cpu loss for maintenance is 52.60410851847717
The percentage of cpu loss for failures is 0.583342637043657
The percentage of memory loss for maintenance is 48.62974522719375
```

ANALYSIS BY GABRIEL ANTHUNES

Analysis of the distribution of machines according to CPU and MEMORY capacity

This analysis can be found in the “machines_distribution_analysis.py”. Its purpose is to use the *machine_events* file to estimate the distribution of machines in relation to its CPU and memory capacity. First, we begin by creating a RDD with the entries of the file, then we change it to a more useful format for us. This is done with the combination of a *map()* method to get (machine ID, (CPU, Memory)) key-value pairs with only the relevant fields for the analysis, and after with the *distinct()* and *filter()* methods to select distinct non-empty entries. With that we are able to calculate the total number of valid machines with:

```
machine_nb = machines.count()
```

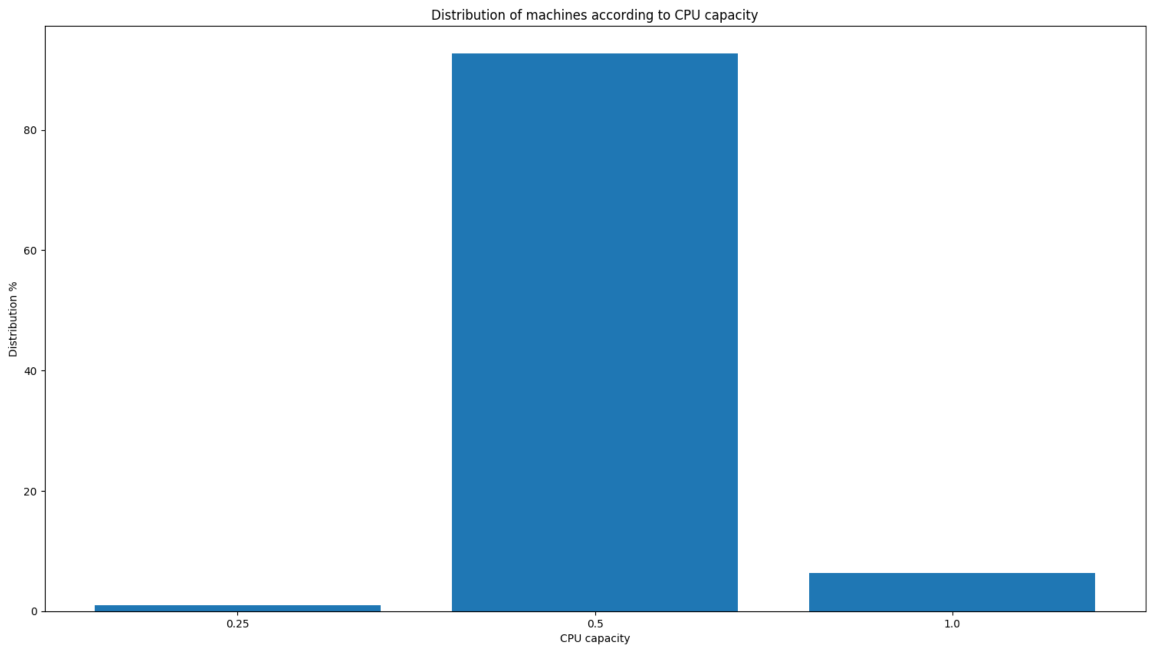
Now we need to count the number of machines for each CPU and memory capacities, to do so we apply another *map()* to the previous RDD to get new key-value pairs in the form of (CPU capacity, 1) and (Memory capacity, 1), and Spark has a counting method for these cases in *countByKey()*(*reduceByKey(add)* was an equally valid option). Finally, we divide these values by the total number of machines to get the distributions. These operations can be seen on the lines:

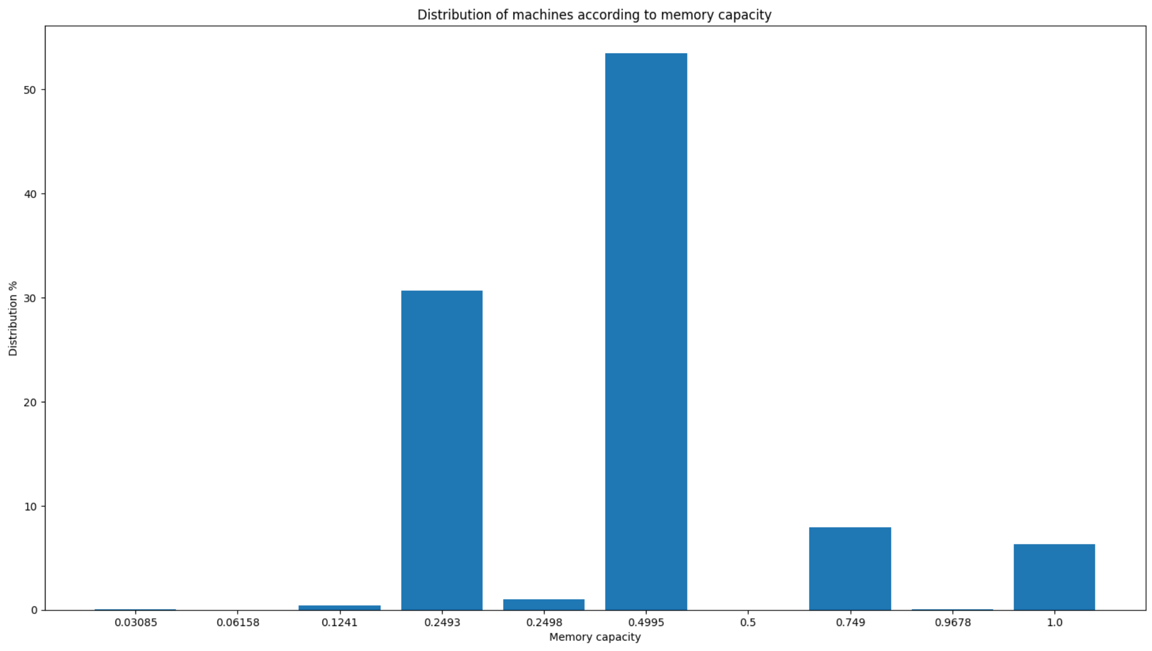
```
dist_cpu = machines.map(lambda x: (float(x[1][0]),1)).reduceByKey(add).map(lambda x: (x[
0], float(x[1])*100/machine_nb))
dist_mem = machines.map(lambda x: (float(x[1][1]),1)).reduceByKey(add).map(lambda x: (x[
0], float(x[1])*100/machine_nb))
```

The output on the terminal looks like this:

```
Distribution of machines according to cpu capacity:

CPU capacity: 0.25; Distribution: 1.0007942811755361%
CPU capacity: 0.5; Distribution: 92.66084193804606%
CPU capacity: 1.0; Distribution: 6.338363780778396%

Distribution of machines according to memory capacity:

Memory capacity: 0.03085; Distibution: 0.03971405877680699%
Memory capacity: 0.06158; Distibution: 0.007942811755361398%
Memory capacity: 0.1241; Distibution: 0.4289118347895155%
Memory capacity: 0.2493; Distibution: 30.706910246227164%
Memory capacity: 0.2498; Distibution: 1.0007942811755361%
Memory capacity: 0.4995; Distibution: 53.47100873709293%
Memory capacity: 0.5; Distibution: 0.023828435266084195%
Memory capacity: 0.749; Distibution: 7.966640190627482%
Memory capacity: 0.9678; Distibution: 0.03971405877680699%
Memory capacity: 1.0; Distibution: 6.314535345512311%
Total time elapsed: 3.038912534713745 seconds.
```

The resulting distributions can be seen on the following plots:





We can easily note that for both the CPU and memory capacities, the trend seems to be that the machines end up disponibilizing half of its resourses, other than that, small variations can occur but not as significant. This analysis was conducted by Spark in **3.04** seconds.

Question: Do tasks with low priority have a higher probability of being evicted?

For this analysis we will use the *task events* file. We are interested in computing the probability of a given task event to be an **eviction event**, in relation to its task's *priority*. That is, in relation to the priority of the job for which this given task belongs to. Indeed, as the documentation (https://drive.google.com/open?id=0B5g07T_gRDg9Z0lsSTEtTWtpOW8&authuser=0) explains, the priority of a job normally determines the priority of its tasks. In other words, all tasks related to a job should have the same priority.

In terms of transformations on our *RDD*, we intend to use Spark to *map* it into a new structure that ideally contains one entry for each different priority level that is present in the data set. Each entry in this new structure will in turn contain two distinct pieces of information:

- The probability of an **eviction event** for a task with this given priority.
- The total number of **all the events** that happened to any task with the same parent Job.

This is necessary because we consider the **Probability of Eviction** as the *number of eviction events for any task belonging to a job divided by the number of the total of events for any task belonging to this same job*. The task indexes individually don't have a great importance because the average would remain the same for the Job as a whole.

To achieve this organization, each entry (task event) in the inital *RDD* will be eventually mapped to the following shape:

```
For each task event T:
  (Priority of T, (ID of T's Job, Event Type))
```

Following, we will construct an *Eviction Rate by Priority* RDD, which will consist of yet a new arrangement:

```
For each task event T:
  If Event Type is Eviction:
    (Priority of T , (1,0))
  Else:
    (Priority of T , (0,1))
```

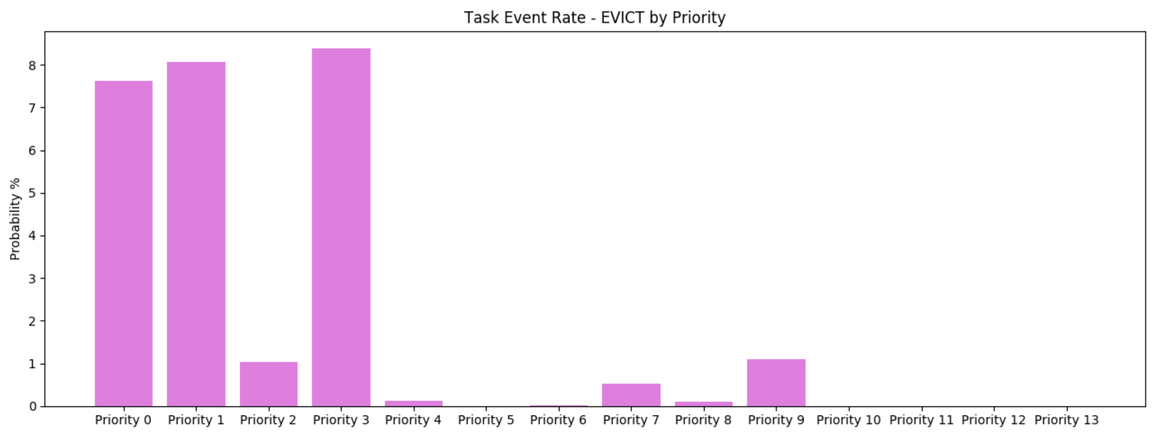
With this new RDD we can know if any given task event was of type *eviction* or not. This is usefull to have a final number of total evictions and of total events for any priority. Finally we will reduce our set by priority key, which means that we will reduce the set to one single entry per priority, adding all this information together and computing the probability at the end. Spark will conduct these operations with commands that looks like this:

```
reduction = evictRate_byPriority.reduceByKey(lambda x, y: (x[0] + y[0], x[1]+y[1])).map(
lambda x: (int(x[0]), (100*(x[1][0]/(x[1][0]+x[1][1])), x[1][0]+x[1][1])))
```

The results are as follows:

```
Probability of evict event by priority:
Priority 0: 7.629659465037097% of having a task evicted. Out of 16875301 events with thi
s priority level.
Priority 1: 8.070885808483835% of having a task evicted. Out of 1741618 events with this
priority level.
Priority 2: 1.0347617678510752% of having a task evicted. Out of 1759922 events with thi
s priority level.
Priority 3: 8.374384236453201% of having a task evicted. Out of 1827 events with this pr
iority level.
Priority 4: 0.1140258749190515% of having a task evicted. Out of 8811158 events with thi
s priority level.
Priority 5: 0.0% of having a task evicted. Out of 93 events with this priority level.
Priority 6: 0.01775301998595539% of having a task evicted. Out of 253478 events with thi
s priority level.
Priority 8: 0.5186483776835573% of having a task evicted. Out of 178541 events with this
priority level.
Priority 9: 0.10400923130988833% of having a task evicted. Out of 3312206 events with th
is priority level.
Priority 10: 1.1040664626147791% of having a task evicted. Out of 9148 events with this
priority level.
Priority 11: 0.0% of having a task evicted. Out of 16025 events with this priority leve
l.

Total time elapsed: 201.9795045852661 seconds.
```

In graphical form, the distribution is as follows:



We can conclude that, indeed, it seems that tasks with low priorities have higher chances of being evicted. This makes sense too, because it is precisely to make available resources to other higher priority tasks that one task may be evicted. We can see that for any given event that happens to lowest priorities, say zero and one, tasks there is around 8% chance of it being an eviction. For the higher priority tasks, starting already from the fourth level, the chance drops to nearly 0% of such an event happening. Finally, out of 16025 events that happened to level 11 priority tasks, none of them were evictions.

This analysis was performed with the use of Spark in **3.36** minutes, processing 32,959,317 lines of data.

Distribution of the number jobs/tasks per scheduling class

We are interested in knowing the distribution of tasks and jobs in relation to the different scheduling classes. This analysis will be done in terms of jobs and in terms of tasks, as such, 100 files from *job_events* as well as 100 files from *task_events* will be used. The procedure will be the same for jobs and tasks, first we will use *spark* to map all the events in terms of Scheduling class and job_id. Following this, we need to eliminate any duplicates because there may be many events for the same task. Finally, we will reduce this newly obtained map by key, which is the scheduling class. This way we can count how many tasks belong to each scheduling class. In the end we will be able to compare it to a total number of tasks to assess the distribution. The base code to cary out these operations looks something like this:

```
allEvents = entries.map(lambda x: (int(x[7]), x[2] )).distinct().map(lambda x: (x[0], 1
))
reduction = allEvents.reduceByKey(add)
```

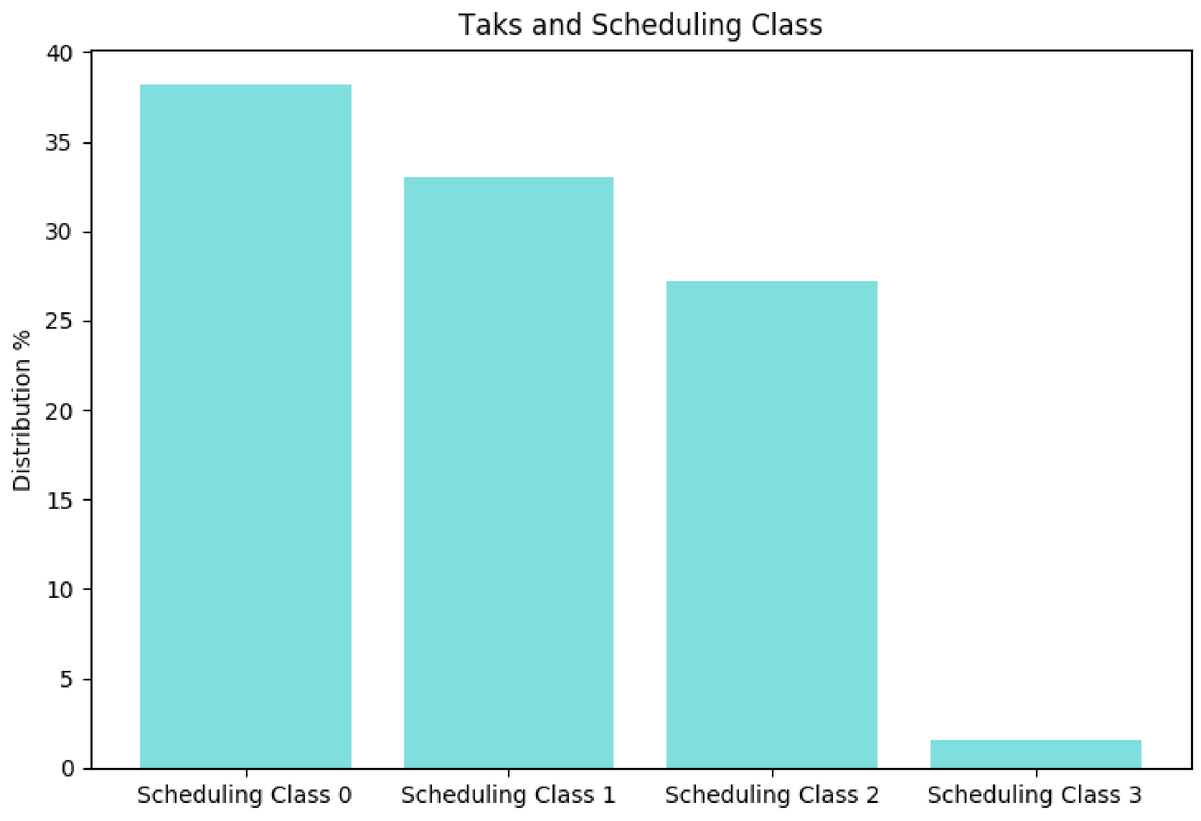
The same procedure for an *RDD* containing job_events:

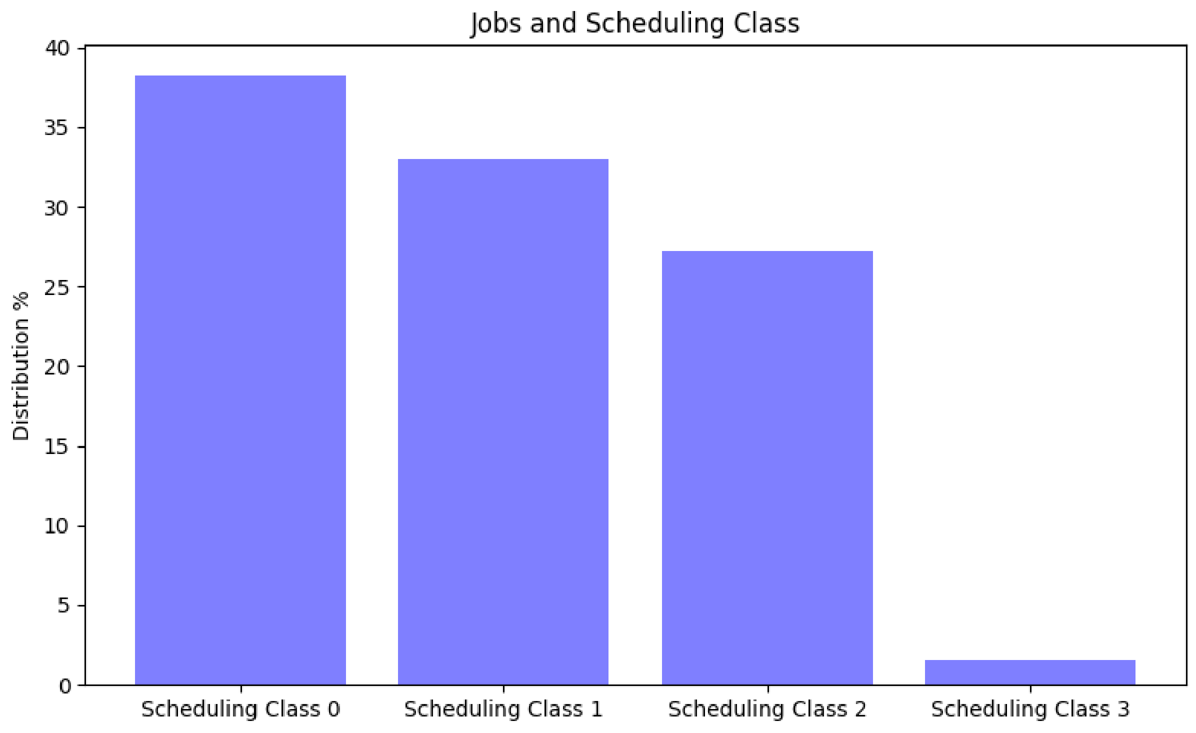
```
allEvents = jobEntries.map(lambda x: (x[5], x[2] )).distinct().map(lambda x: (x[0], 1))
reduction2 = allEvents.reduceByKey(add)
```

The final distribution is as follows:

```
For the tasks:
Scheduling class 0 : 38.23496142795737 percent.
Scheduling class 1 : 33.002284912788724 percent.
Scheduling class 2 : 27.216101471242528 percent.
Scheduling class 3 : 1.546652188011372 percent.
total is 132609
Total time elapsed: 221.31702589988708 seconds.

For the jobs:
Scheduling class 0 : 38.23445156426687 percent.
Scheduling class 1 : 32.99811534112326 percent.
Scheduling class 2 : 27.215981907274784 percent.
Scheduling class 3 : 1.5514511873350922 percent.
total is 132650
Total time elapsed: 45.25424313545227 seconds.
```

We can equally plot this distribution for visualisation purposes:





As expected, we can see that the distributions ammount to 100% together, which is expected. It is also worthwile to mention that the distribution for jobs by scheduling class is very similar to that of the tasks by scheduling class. This is also expected because tasks have the same scheduling class as their parent jobs. Indeed, according to the *google documentation* (https://drive.google.com/open?id=0B5g07T_gRDg9Z0lsSTEtTWtpOW8&authuser=0),jobs normally consist of identic tasks, that demand the same ammount of resources e possess the same priorities and scheduling class. Moreover, the files analyzed correspond to the same time frame which would mean that jobs and their related tasks would be activated within this window.

This analysis was computed using *spark* in **3.68 minutes** for the tasks and **45 seconds** for the jobs, processing 200 files which contain approximately 33,353,310 lines of data. Once again, the transformations *map* and *reduce* came very useful to conduct the operations necessary on the dataset in a conscise way.

Percentage/Frequency of jobs/tasks that got killed or evicted depending on the scheduling class

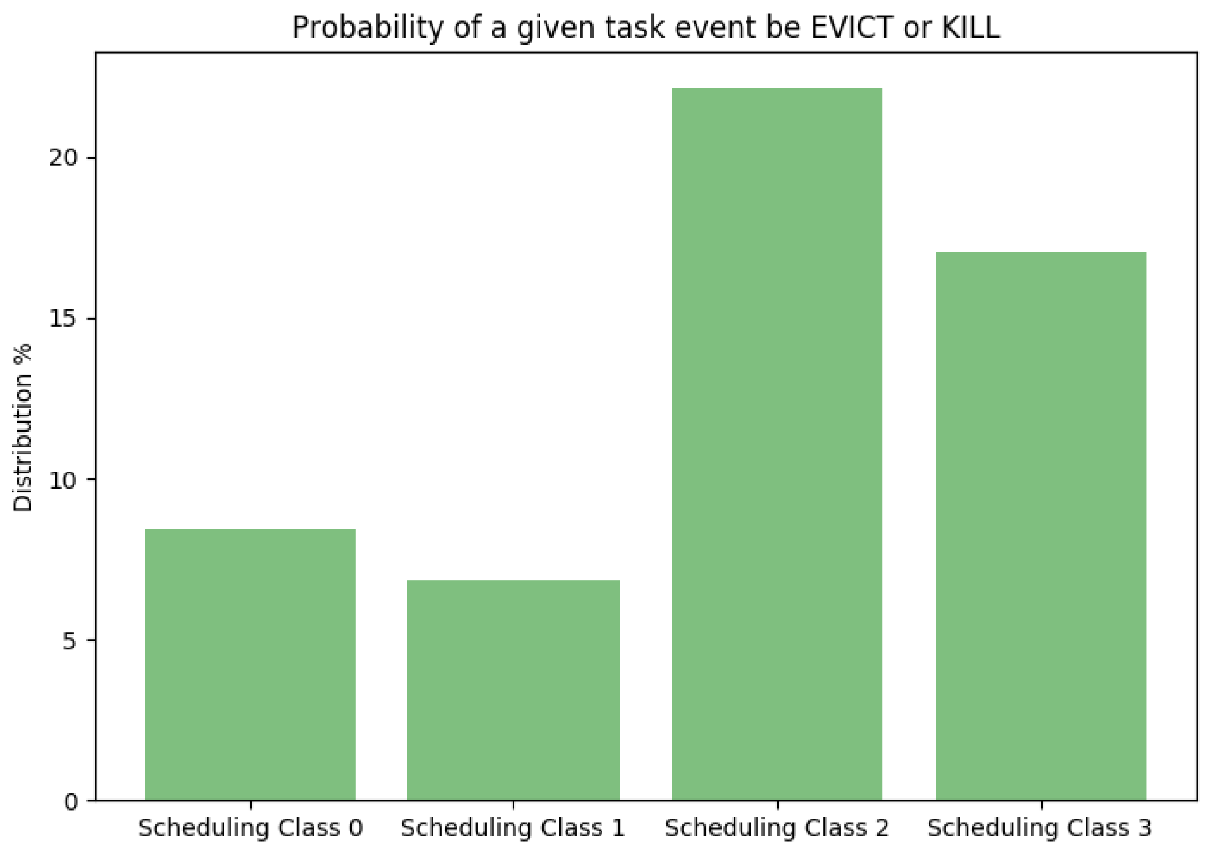
For this question, we will have two approaches to try and measure this relation.

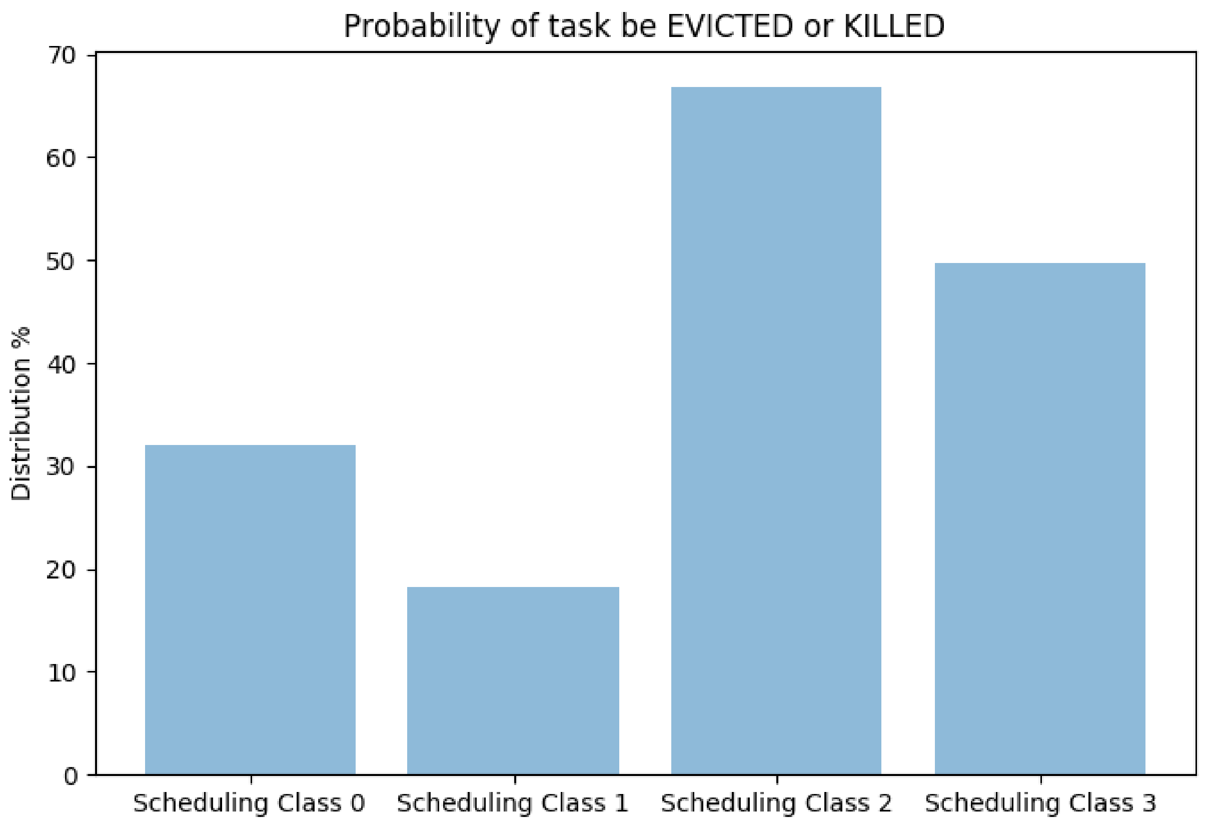
- Firstly, we will split the task events into two groups. The first group contains all events that correspond to a *kill or evict* action. The second group will be all events, including *kill or evict*. We can then organize each event by scheduling class and finally reduce them to have a final relation of *kill or evict* events by total events for any given sheduling class. This will hopefully allow us to estimate, for any given event, what is the probabilty that it will be an eviction or a kill event, based on the scheduling class of this task.
- Secondly, we will try to change the point of view a little bit. Given that a task has a specific scheduling class, what is the chance that this task will be evicted or killed at somepoint. What is the chance that it will not be killed nor evicted? In order to do so, we'll again reduce the number of total events by scheduling class. Following, we'll do the same, except that we'll count the total number with a filtered *RDD* containing only *kill or evict* events. This will enable us to count distinct tasks and see how many were killed or evicted at some point at least once.

The basic transformations are:

```
#First proposition
allEvents = entries.map(lambda x: (x[7], (1,1)) if (x[5] == '2' or x[5] == '5') else (x[
7], (0,1))).reduceByKey(lambda x, y: (x[0] + y[0], x[1]+y[1]))
percentages = allEvents.map(lambda x: (x[0], (100*(x[1][0]/(x[1][0]+x[1][1])))))

#Second proposition
allEvents = entries.map(lambda x: (x[7], (1, (x[2], x[3])))).distinct().reduceByKey(lamb
da x, y: (x[0] + y[0], x[1]))
tasksThatWereEvictedOrKilled = allEvents.filter(lambda x: x[5] == u'2' or x[5] ==u'5').m
ap(lambda x: (x[7], (1, (x[2], x[3])))).distinct().reduceByKey(lambda x, y: (x[0] + y[0
], x[1]))
```

After manipulating the results to calculate averages, they are printed on the output terminal. We can then generate the following graphics:


Probability of a given task event be EVICT or KILL
Distribution %
0
5
10
15
20
Scheduling Class 0
Scheduling Class 1
Scheduling Class 2
Scheduling Class 3


Probability of task be EVICTED or KILLED
Distribution %
0
10
20
30
40
50
60
70
Scheduling Class 0
Scheduling Class 1
Scheduling Class 2
Scheduling Class 3

First Step

```
Probability of task event being EVICT or KILL based on scheduling class:
('0', 8.422286770195647)
('1', 6.850361433152736)
('2', 22.141683206075324)


('3', 17.02527194943253)
First part finished. Time elapsed: 191.8826973438263 seconds.
```

Second Step

```
Percentage analysis
Scheduling class 0 :32.03944801867268 percent.
Scheduling class 1 :18.265014034833516 percent.
Scheduling class 2 :66.9128349599222 percent.
Scheduling class 3 :49.67025745734469 percent.
Total time elapsed: 564.0192849636078 seconds.
```

These computations, using *map*, *reduce* and *filter* operations, were conducted by *Spark* in **3.2 minutes** for the first step and **9.40 minutes** for the second step. Over this time, spark was processing around 32,959,317 lines of data, from the first 100 *task_event* files available in the dataset, which is quite impressive.

We can see in the first graph that there isn't a very strong relation between a scheduling class of a task, and the frequency with which they can be evicted or killed. In general it seems that scheduling classes 0 and 1 have tasks which can be evicted or killed more often then scheduling classes 2 and 3. In the second graph the distribution at first looks similar, tasks which have scheduling classes 1 or 2, that are evicted or killed at least once, are less frequent then in the other scheduling classes. This time however, Scheduling classes 2 and 3 have tasks that are much more prone to being evicted or killed in their lifetime.